Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 69 (2007) 14–26
Science of Computer Programming
www.elsevier.com/locate/scico

The JastAdd system — modular extensible compiler construction

Torbjörn Ekman*, Görel Hedin

Department of Computer Science, Lund University, Sweden

Received 1 October 2005; received in revised form 30 July 2006; accepted 8 February 2007
Available online 10 October 2007

Abstract

The JastAdd system enables modular specifications of extensible compiler tools and languages. Java has been extended with the Rewritable Circular Reference Attributed Grammars formalism that supports modularization and extensibility through several synergistic mechanisms. Object-orientation and static aspect-oriented programming are combined with declarative attributes and context-dependent rewrites to allow highly modular specifications. The techniques have been verified by implementing a full Java 1.4 compiler with modular extensions for non-null types and Java 5 features.

Keywords: Compiler construction; Extensible languages; Modular implementation

1. Introduction

We present the JastAdd system for the development of compilers and related tools. The system enables modular specification of extensible compiler tools and languages. JastAdd is an extension to Java that supports a specification formalism called Rewritable Circular Reference Attributed Grammars (ReCRAGs). ReCRAGs raises the abstraction level for grammar-based computations using declarative object-oriented techniques to rewrite abstract syntax trees (ASTs).

Several synergistic mechanisms are combined to better support modularity and extensibility: object-orientation, static aspects, declarative attributes, and context-dependent rewrites. The combination of these mechanisms allows the compilation problem to be decomposed into almost any way, according to the tool writer's choice. For example, the modules can be chosen to match the chapters of a language documentation. A set of modules that defines a base language can be reused by other module sets that extend the base language. A set of modules that defines the static-semantic analysis for a language can be reused for many different tools for that language, e.g., a compiler, a source metrics tool, etc. To illustrate the power of this technique, we have used JastAdd to implement a modular Java 1.4 compiler [1], and several extensions to this compiler. One extension is a compiler for large parts of Java 5. Another is a tool that adds non-null types to Java and infers non-null properties in legacy library code. A third extension is an

* Corresponding address: Oxford University Computing Laboratory, Wolfson Building, Parks Road, OX1 3QD Oxford, United Kingdom. Tel.: +44 1865 283 512; fax: +44 1865 273 839.
E-mail addresses: torbjorn@cs.lth.se, torbjorn.ekman@comlab.ox.ac.uk (T. Ekman), gorel@cs.lth.se (G. Hedin).
URL: http://jastadd.cs.lth.se (T. Ekman, G. Hedin).

0167-6423/$ - see front matter
doi:10.1016/j.scico.2007.02.003

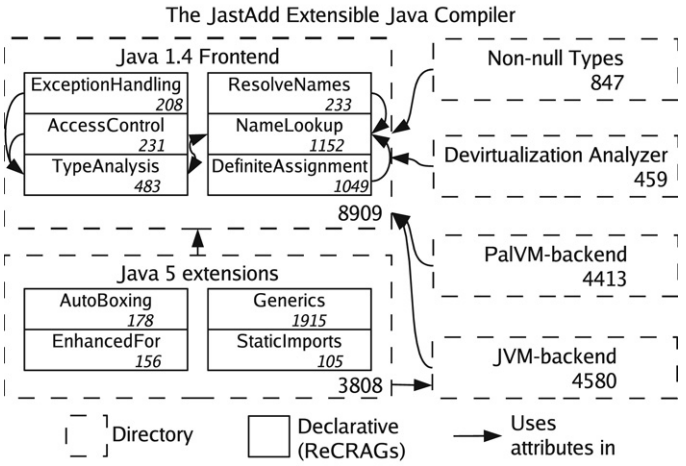

Fig. 1. The JastAdd Extensible Java Compiler with various extensions. The numbers represent lines of code (SLOCCount).

experimental tool that computes cross-references and devirtualization metrics, reusing the static-semantic analysis of the Java 1.4 compiler.

A major reason why the modules can be designed and extended so flexibly is the declarativeness of the ReCRAG mechanisms. The order in which the attribute computations and rewrites are carried out is decided automatically through their implicit dependences instead of requiring explicit ordering in the code.

We have verified that the techniques scale to real languages by validating the generated Java 1.4 compiler against the Jacks test suite [2], and it passes more tests than both the popular javac and jikes compilers. The size of the specification is less than two-thirds the size of the handwritten javac implementation and the generated compiler's performance is within a factor of three.

The integration of the declarative attributes and rewriting techniques with Java and its well-known object-oriented features, makes the system fairly easy to learn for software developers in the industry, thus not restricting its use to the computer science research community. To use the system, no previous knowledge of attributes grammars, aspects or rewrite systems is necessary. The documentation explains these features in relation to ordinary object-oriented programming.

The rest of this paper is structured as follows. Section 2 introduces the intended application domain through a case study of a full Java compiler implemented in JastAdd. Section 3 gives an overview of the architecture. Section 4 describes how various features can be used to specify modular and extensible descriptions. Section 5 discusses error checking of the specifications, and Section 6 briefly explains the evaluation method used. Section 7 compares our system to related tools and systems. Section 8 concludes the paper.

2. Application domain

The primary application domain for the JastAdd system is extensible compilers and analysis tools. It has been used to implement full languages like Java, domain-specific languages for robotics and automation, and numerous smaller toy languages in an undergraduate course. The rest of this section uses the JastAdd Extensible Java Compiler to illustrate the application domain, but the distribution contains several smaller examples.

The basis for the compiler is a front-end that performs static-semantic analysis for Java 1.4. This front-end is modularized according to the various concerns in the analysis, e.g., name resolution, type checking, definite assignment, unreachable statements, and exception handling. It can be used as a stand-alone tool and has also been extended with various experimental analysis modules such as devirtualization and metrics. The front-end is used by multiple back-ends that target different runtime environments [3]: byte code for Java Virtual Machines, byte code for the Palcom Runtime Environment [4], and C source code. Fig. 1 shows a schematic view of the Java 1.4 compiler and various extensions.

The initial decomposition criterion for the Java 1.4 compiler was functionality, but we have also extended it with new non-trivial language constructs that cross-cut existing modules. An experimental extension to large parts of Java 5

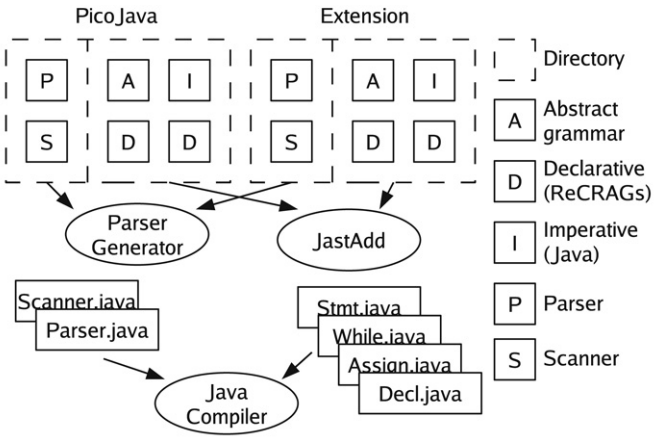

Fig. 2. Generation architecture — describes the generation of plain Java source files from the various specifications.

has been implemented, including complex language features like generics with wildcards that extend the type system significantly. This extension is ongoing work and is not yet up to the same high quality as the Java 1.4 compiler. But the main challenges in Java 5 have already been solved and only details and extensive testing remains. Another extension adds non-null types (as described in [5]) to Java, also a significant extension to the type system. A third extension adds AspectJ-style inter-type declarations that include highly non-local changes to name resolution. All the extensions were implemented in a completely modular, yet understandable way.

3. Architecture

A main design idea behind JastAdd is to make ReCRAGs easily accessible to Java programmers. For this reason, JastAdd uses a syntax very close to Java, only extended with constructs to support the features of ReCRAGs such as attributes, equations, and rewrites. ASTs are represented by Java classes and attributes by Java methods.

A typical generated tool consists of a small main program that first invokes a parser to build the AST and then invokes a small imperative module to output the desired results, e.g., writing the compiled code to a file. The bulk of the computations is performed by the declarative ReCRAG modules that define the desired properties of the AST as attributes, ultimately accessed by the imperative module that outputs the results. The actual evaluation of the attributes is done automatically as a result of accessing them.

The source code for a generated tool is organized in a directory containing abstract grammar modules, declarative ReCRAG modules, and imperative Java modules. JastAdd takes a set of modules as input and generates ordinary Java classes for the AST, according to the abstract grammar modules. These generated classes contain woven generated code that implement the behavior specified in the ReCRAG modules. Fig. 2 gives a schematic view of the generation from ReCRAG specifications to plain Java source files.

To build an extended tool, an additional directory for the extension is created, adding new abstract grammar modules, declarative ReCRAG modules, and imperative modules as desired. The extended tool is generated by providing JastAdd with modules from all the desired directories as input. There is no specific module combination language — the modules are simply listed as input to JastAdd, typically in a build script.

The central program representation in JastAdd is the AST which is defined by an abstract grammar such as the one shown in Fig. 3. Each production defines a new Java class with an optional superclass (written as ‘:’ and a superclass name). The right-hand side of a production defines the children of that node, and the list of children is appended to children inherited from the superclass. Accessors are generated for the children and they are also included as parameters to node constructors. Each child can optionally be named by prefixing it with a name followed by ‘:’, and otherwise the name is the same as the type of the child. A child enclosed in ‘<’ and ‘>’ is a lexeme of a primitive type rather than a tree node. ‘[’ and ‘]’ indicate that the child is optional while ‘*’ declares a list of children. A child enclosed in slashes is a non-terminal attribute, see Section 4.6.

Support for parsing is not provided by JastAdd, but any Java-based parser generator can be used, e.g., JavaCC, ANTLR, CUP, or Beaver. The goal of the parsing is to build an AST that follows a JastAdd abstract grammar. Fig. 4

```
Program ::= Block /ObjectType:ClassDecl/;
abstract Stmt;
Block : Stmt ::= Stmt*;
abstract Decl : Stmt ::= <Name:String>;
ClassDecl : Decl ::= [Superclass:Access] Body:Block;
VarDecl : Decl ::= Type:Access;
AssignStmt : Stmt ::= Variable:Access Value:Exp;
abstract Exp;
abstract Access : Exp;
abstract Use : Access ::= <Name:String>;
QualUse : Access ::= ObjectReference:Access Use;
```

Fig. 3. A JastAdd abstract grammar.

```
BlockStmt block-stmt = class-decl | var-decl | stmt ;
Stmt stmt = assign-stmt | ... ;
ClassDecl class-decl = CLASS ID extends-opt block
{: return new ClassDecl(ID, extends_opt, block); :};
VarDecl var-decl = name ID SEMICOLON
{: return new VarDecl(ID, name); :};
AssignStmt assign-stmt = name ASSIGN exp SEMICOLON
{: return new AssignStmt(name, exp); :};
Exp exp = name | ... ;
Access name = ID {: return new Use(ID); :}
 | name DOT ID {: return new QualUse(name, new Use(ID)); :};
```

Fig. 4. A parser specification that builds AST nodes in its semantic actions.

shows some typical BNF parser rules and how the AST is built in the semantic actions of the parser, using the node constructors as defined by the abstract grammar in Fig. 3.

4. Specification features

In JastAdd, the AST is modelled as an object-oriented class hierarchy that allows behavior to be defined in classes and specialized in subclasses. In combination with inter-type declarations [6] (also known as open classes [7]), computations that cross-cut the class hierarchy can be defined in separate modules, thus enabling good separation of concerns. To specify behavior, declarative attributes are used rather than imperative codes. The declarative approach supports a high degree of decoupling since the order of evaluation does not have to be explicitly defined. JastAdd also supports context-dependent rewrites which allows the AST to be dynamically changed to better suit different computations, thereby improving the modularization of many computations. In this section we briefly describe the different features in JastAdd and exemplify their use. The examples are based on the language defined in the previous section, but the same techniques are used in our extensible Java compiler.

4.1. Attributes

Attribute Grammars (AG) [8] is a technique for defining computations on an AST in a declarative way and integrates well with the object-oriented programming paradigm [9,10]. Attributes are declared for nodes in the AST and their values are defined by equations that may use the values of other attributes in the tree. A *synthesized* attributes is defined by an equation in the node itself whereas an *inherited* attribute is defined by an equation in an ancestor node.

Synthesized attributes decouples the specification of an attribute from its actual implementation. Consider the code in Fig. 5 that defines a synthesized attribute `value()` that denotes the value of an expression. The attribute is declared for all `Expr` nodes and an equation is provided for each kind of expression. JastAdd weaves attributes and equations into the classes generated from the abstract grammar. The generated classes contain `get`-methods that are used in equations to access node children and lexemes. Synthesized attributes are, in the object-oriented notation for AGs, very similar to traditional virtual methods in object-oriented programming. The main difference is that synthesized attributes must be functions that may not have side effects. This allows for efficient evaluation through caching of the result. That same result is then returned for each occurrence of that attribute in other equations.

```
abstract Expr;
Literal : Expr ::= <Value:String>;
AddExpr : Expr ::= Left:Expr Right:Expr;
SubExpr : Expr ::= Left:Expr Right:Expr;

syn int Expr.value();
eq Literal.value() = Integer.parseInt(getValue());
eq AddExpr.value() = getLeft().value() + getRight().value();
eq SubExpr.value() = getLeft().value() - getRight().value();
```

Fig. 5. Computing expression values.

```
WhileStmt : Stmt ::= Cond:Exp Stmt;
AssignStmt : Stmt ::= LValue:Exp RValue:Exp;

inh Type Exp.expectedType();
eq WhileStmt.getCond().expectedType() = booleanType();
eq AddExpr.getLeft().expectedType() = intType();
eq AddExpr.getRight().expectedType() = intType();
eq AssignStmt.getLValue().expectedType() = anyType();
eq AssignStmt.getRValue().expectedType() = getLValue().type();
```

Fig. 6. Computing the expected type for expressions.

```
aspect TypeComputation {
  syn ClassDecl Decl.type();
  syn ClassDecl Exp.type();
  eq ClassDecl.type() = this;            // (1)
  eq Use.type() = decl().type();         // (2)
  eq QualUse.type() = getUse().type();   // (3)
  eq VarDecl.type() = getType().type(); // (4)
}
```

Fig. 7. Type computation.

Inherited attributes decouples an AST node from its parent: the AST node does not need to know which parent it has. All the information it needs is in the inherited attributes whose values are defined by the parent. This allows AST classes with all their behavior to be reused in many different contexts. Consider the code in Fig. 6 that defines an attribute `expectedType()` that denotes the expected type of an expression used during type checking to find ill-typed expressions. Expressions occur in many different statements and declarations and the valid type of an expression depends on its context. The while statement requires its condition to be of boolean type. Addition only operates on integer-typed values and provides equations for both its children. Assignment requires the right-hand side to have the same type as the left-hand side, while the left-hand side may have any type. An expression need not know which parent it has during type checking but can simply compare the `expectedType()` attribute to its actual type.

4.2. *References add additional structure to the AST*

Reference Attributed Grammars (RAGs) [11–14] allow attributes to be references to other tree nodes. This allows graphs to be represented on top of the AST, something that is very useful in compilers and similar tools. References can be used to model various language relations such as name bindings. i.e., a use of a name is bound through a reference to its corresponding declaration. Types can, in a similar way, be represented by their declarations and then the type of an expression is simply represented by a reference from the expression node to a type declaration node. Consider the computation of the type for declarations and expressions in Fig. 7 (please refer to Fig. 3 for the abstract grammar that this computation applies to). The `type()` attribute is a reference to a `ClassDecl` node which represents a type. The type of a `ClassDecl` is the declaration itself (1). A simple name `Use` is bound to a declaration, and the type of the `Use` is the same as the type of the declaration (2). Notice that this equation depends on the name binding module defining the `decl()` attribute (a reference from each `Use` to its corresponding declaration). The type of a qualified name `QualUse` is the the same as the type of its last name (3). Finally, the type of a `VarDecl` is the same as the type of its `Type` child (4).

```
aspect NameBinding {
  syn Decl Use.decl() = lookup(getName());
  inh Decl Use.lookup(String name);
  inh Decl Block.lookup(String name);
  eq Block.getStmt().lookup(String name) {
    // First, look in local declarations
    if(localLookup(name) != null)
      return localLookup(name);
    // Then, enclosing
    return lookup(name);
  }
  syn Decl Block.localLookup(String name) {
    // Find the first statement that declares name
    for (int k = 0; k < getNumStmt(); k++)
      if(getStmt(k).declarationOf(name) != null)
        return getStmt(k).declarationOf(name);
    return null;
  }
  // Return a statement that declares name or null
  syn Decl Stmt.declarationOf(String name) = null;
  eq Decl.declarationOf(String name) {
    if(getName().equals(name))
      return this;
    return null;
  }
}
```

Fig. 8. Name binding. Each `Use` is bound to a visible declaration. Blocks provide nested scopes with shadowing.

The use of reference attributes allows the AST to be used as the only data structure in the compiler. This is in contrast to the usual approach using symbol tables: instead of encoding all information about variables and types in a complex separate symbol table, the AST nodes for the variable and type declarations are used directly. Lookup in the symbol table is replaced by a reference attribute defined using a lookup attribute, see the section on broadcasting below. In addition to name bindings, references are used extensively in the Java compiler to model various graph structures such as the inheritance hierarchy, recursive types, and the call graphs. Relations can also be modelled as sets of references. For example, a method overrides/hides a set of other methods; a type is a subtype of a set of types.

4.3. Broadcasting scope and parameterized attributes

Inherited attributes decouple the AST node from its parent. This can be improved further by decoupling the parent from its child. Inherited attributes can be broadcasted not only to a single child but to all its descendants. This feature is similar to the *including* feature in the Eli system [15]. The scope for the attribute equation is then widened from a single node to an entire subtree. The only requirement for an inherited attribute to be well-defined is then that there is a path from the declaration of an attribute through its ancestors to an equation. By allowing a descendant to redefine the equation for its subtree, the broadcasting mechanism is an excellent way to describe nested scopes. It is highly desirable from a modularization point of view that the modules defining scoped data need not be aware of the modules using the scoped data and vice versa. This makes it easy to extend the language with new constructs, both constructs that introduce new scopes, and constructs that make use of scoped data.

This loose coupling can be further enhanced by allowing attributes to have parameters. For example, the set of visible variables can be parameterized by a name. Information can then flow both upwards, through the argument, and downwards, by the broadcasted attribute, while keeping the modularization benefits described above. Consider the name binding module in Fig. 8. The external interface is an attribute `decl()` that binds each `Use` to a declaration, which is implemented internally by a parameterized attribute `lookup(String name)` that binds a name in the current context to a visible declaration. Each `Block` defines a new scope and provides an equation that first searches for local declarations, and then delegates to the enclosing context, i.e., a nested block, if no local declaration is found. This implements nested scopes with shadowing. The `localLookup(String name)` attribute iterates over the statements in the block and returns the first found declaration matching a certain name.

```
aspect InheritedAndRemoteMemberLookup {
  eq ClassDecl.getBody().lookup(String name) {
    // First, look in inherited members
    if(memberLookup(name) != null)
      return memberLookup(name);
    // Then, enclosing
    return lookup(name);
  }
  // Qualified lookup of members
  eq QualUse.getUse().lookup(String name) =
    getObjectReference().type().memberLookup(name);

  syn Decl ClassDecl.memberLookup(String name) {
    // First, lookup local members
    if(getBody().localLookup(name) != null )
      return getBody().localLookup(name);
    // Then, superclass chain
    if(superClass() != null)
      return superClass().memberLookup(name);
    return null;
  }
  syn ClassDecl ClassDecl.superClass() =
    hasSuperclass() ? getSuperclass().type() : null ;
}
```

Fig. 9. Name binding for inherited and remote members.

4.4. Complex non-local dependences

The superimposed graph structures on top of the AST, in combination with declarative attributes, makes it easy to decompose a complex computation into several simpler ones. A combination of broadcasting of parameterized attributes and delegation through a reference allows complex, highly non-local, dependences to be expressed is a simple fashion. Object-oriented inheritance can, for instance, be implemented by using broadcasting for nested scopes combined with delegation to the superclass through a reference. Consider the code in Fig. 9, implementing lookup for inherited members as well as for remote access to members. The memberLookup(String name) attribute includes not only local declarations but delegates the search to its superclass if no local declaration is found. This implementation matches traditional object-oriented inheritance with overriding. If the member lookup is fruitless the lookup is delegated to the enclosing broadcast in the same way as for nested blocks. This implementation matches nested classes. Qualified names, e.g., `a.b.c`, are used to remotely access members in a referenced object. The implementation delegates the member lookup to the type declaration matching the type of the `ObjectReference` child in the qualified name `QualUse`.

4.5. Enhance tree structure through rewrites

The initial context-free AST, often built by a parser, is not a perfect match for most computations. Since the AST is the only data structure in the system it would be quite limiting if the AST could not be changed. *Rewrites* allow the initial AST to be rewritten to a more suitable form, based on values of the context-dependent attributes. This allows the tree to not only reflect context-free structure but also the context-sensitive information that has been computed so far. For example, a name node can be rewritten to a specialized node to reflect its semantic meaning, like a variable- or type-name, as soon as that information is available. This allows later computations, like optimization and code generation, to be modularized according to the semantic meaning of the name. For example, splitting the generation of code for variable accesses and type-name accesses into separate modules. Rewrites can thus be used to make the tree more suitable for the other modularization features in JastAdd. The example in Fig. 10 rewrites context-free Use nodes into context-sensitive ones reflecting whether the corresponding declaration is a variable or a type declaration.

The rewriting is done iteratively, interleaved with attribute evaluation. This allows complex rewrites to be broken down into many simple rewrites. The fine-grained interaction between rewrites and attribute evaluation is further discussed in [16].

```
aspect SemanticSpecialization {
  rewrite Use {
    when(decl() instanceof VarDecl)
    to VariableUse new VariableUse(getName());
  }
  rewrite Use {
    when(decl() instanceof ClassDecl)
    to TypeUse new TypeUse(getName());
  }
}
```

Fig. 10. Rewrite rules for replacing context-free use nodes with context-sensitive ones based on their declaration kind.

```
aspect PredefinedObjectType {
  // Program ::= Block /ObjectType:ClassDecl/;
  syn ClassDecl Program.getObjectType() =
    new ClassDecl("Object", new Opt(), new Block());
  eq Program.getBlock().lookup(String name) =
    getObjectType().declarationOf(name);
}
```

Fig. 11. Non-terminal attribute that adds the predefined `Object` class.

Rewrites can also be used to normalize the AST into a kernel language. This can be used for rewriting language extensions to constructs in the base language, something which often requires contextual information. Yet another use of context-dependent rewrites is to separate language constructs that are ambiguous from a context-free perspective, such as the infamous typedef vs. variable ambiguity in C.

The Java compiler uses rewrites to make implicit language behavior explicit in the AST. For example, to add default constructors to classes with no constructors, and to add an explicit *this* to unqualified invocations of instance methods. A series of rewrites are also used to resolve syntactically ambiguous names, as described in more detail in [17]. Other uses of rewrites include splitting variable declarations with multiple variables and grouping scattered cardinality for array names.

4.6. Non-terminal attributes

It is sometimes useful to expand an AST with additional nodes that are defined by equations, rather than constructed by the parser or by a rewrite rule. These nodes are called non-terminal attributes (NTAs) [18,19] since they are both similar to nodes (non-terminals) and to attributes. An NTA is like a node in that it can itself have attributes and it can be rewritten. It is also like an attribute in that it is defined by an equation. The code in Fig. 11 implements the non-terminal attribute `ObjectType` in the `Program ::= Block /ObjectType:ClassDecl/;` production in the abstract grammar. The predefined `Object` class is built and added to the AST through an NTA. Notice that this equation may read other attributes in the AST even though the feature is not used in the example. The NTA attribute is then used in a `lookup(String)` equation to make the declaration visible to the name binding module in Fig. 8.

Grammars with NTAs are considered higher-order attribute grammars since attributes may themselves have attributes. There are similarities between NTAs and context-dependent rewrites. In both cases, you can declaratively define changes to the AST based on attribute values. The main difference is that you should use rewrites when you are interested in replacing some nodes with others, and NTAs when you want to keep existing nodes and introduce some additional ones. NTAs in combination with a technique called attribute forwarding [20] would be similar to replacing a tree by a new one.

NTAs are used in the Java compiler to add predefined declarations, such as primitive types, but more importantly to instantiate generic types in Java 5. The NTA is then a function of the generic type declaration parameterized with one or more type parameters.

4.7. Circular attributes

JastAdd supports circular attributes [21] which are useful for many analysis problems, for example in code optimization. Attributes may then be mutually dependent and the evaluator computes fixed-point solutions by iteration.

```
aspect CircularClassHierarchies {
  // Circular attribute with bottom value true
  syn boolean ClassDecl.circularHierarchy() circular [true];
  eq ClassDecl.circularHierarchy() {
    // First, check if there is a superclass
    // If so, this class is circular if its superclass is
    if(superClass() != null)
      return superClass().circularHierarchy();
    return false;
  }
}
```

Fig. 12. Detection of cycle on the superclass chain.

The use of circular attributes is particularly useful in combination with reference attributes, since they can then be used to compute mutually recursive properties on top of graphs [22,13].

In order to guarantee termination of the circular attribute evaluation, the usual lattice approach is used: the possible values of circular attributes are arranged in a lattice of finite height. The bottom value is used as the start value and all equations should be *monotonic*, i.e., when the equation is used as an assignment by the evaluator, it can only result in the same value or a value higher up in the lattice. Because the lattice height is finite, the evaluation will terminate. In practice, we have used boolean lattices and set lattices. A boolean lattice simply consists of the values true and false, for example selecting true as bottom and false as top. A typical set lattice has the empty set at the bottom and the set of all items of some kind at the top, for example the set of all identifiers in the program.

The code in Fig. 12 shows a simple example that detects cycles in class hierarchies. The lattice is a boolean lattice with true as bottom and false as top. A circularity occurs if a class reaches itself by following its transitive closure of superclasses. A class that does not have a superclass returns false which is the lattice top. We have used circular attributes in the Java compiler to detect circularities in inheritance hierarchies and to determine definite assignment properties of variables in loop constructs.

4.8. Imperative modules

Aspect modules with inter-type declarations can be used not only for declarative attributes but also for imperative Java code. Methods and fields can then be introduced in existing classes in a modular fashion. Imperative code may use declarative attributes by invoking them as if they were methods. Attribute evaluation and rewriting of the AST is transparent to the imperative code.

Imperative modules are useful for outputting results to files. For example, in the Java 1.4 front-end, name resolution and type checking is done completely by declarative modules, but the output of compile-time errors is done by a small imperative module which traverses the AST and uses the attributes to print out suitable error messages. The Java 1.4 back-end includes an imperative module that traverses the AST and uses the attributes to output the tree to a bytecode file.

Another use of imperative modules in the Java 1.4 compiler is to implement demand-driven parsing of bytecode files. Conceptually, all Java classes are available in a large AST. But for efficiency, the AST subtrees for the imported classes are built on demand, by parsing bytecode files as the imported classes are accessed during compilation.

5. Specification error detection

JastAdd includes some simple static and dynamic checks of the specifications. It is checked statically that the attributes are well-defined, i.e., each attribute will have a defining equation for any possible AST. It is checked dynamically that ordinary attributes (not declared as circular) are not in fact circularly defined. While plain AGs can use a static circularity test [8], reference attributes make static determination of circularity undecidable [14], hence the dynamic test.

There are several typical kinds of specification errors that can occur, but that are currently not detected by the system. It is future work to add static or dynamic checks to be able to detect some of these errors. One example is to detect if rewrites terminate. Non-terminating rewriting can occur, for example, if a condition is written too broadly so that it remains true even after the rewrite. For this simple case, a dynamic test can probably be added, but there are

more intricate cases which are more difficult to test. Another issue is confluency. If two rewrites are applicable at the same time, it is desirable that they are confluent, i.e., the order of applying them is irrelevant. In the current system, the evaluator will select between applicable rewrites depending on lexical order. It is not checked if another evaluation order would give a different result. If it would, the conditions should probably be constrained so that only one of the rewrites was applicable at a time, thus ordering the rewrites without having to rely on brittle lexical order. However, in our experience, nonconfluency is seldom a problem in practice, because rewrites that are applicable at the same time usually rewrite unrelated parts.

An additional issue is the possibility of interaction between rewrites and circular attributes. Our evaluation algorithm assumes that a circular attribute is not evaluated until all the involved nodes are already rewritten, but this is currently not checked. In our current specifications, this has not been any problem, but we plan to add dynamic checks to the evaluator to signal if an interaction should occur. Further research is needed to investigate if there are practical situations where mixed evaluation of rewrites and circular attributes would be useful, and to generalize the evaluation algorithm.

It would also be useful to add non-null types [5] in order to be able to statically avoid dereferencing of null references and to add static checks that ensure that equations do not have external side effects. An additional improvement would be to add lattice types for use in circular attributes as in [13]. Currently the system does not check that the equations for circular attributes are indeed monotonic, and failing to write monotonic equations may result in non-terminating evaluation.

6. The JastAdd evaluation engine

The evaluation algorithm in JastAdd is built on lazy dynamic evaluation with attribute caching. Dependences are computed dynamically and on demand. Since references may reference arbitrary nodes it is difficult to schedule the order statically. Some research exists on doing this [14], but this results in conservative approximations, and the applicability and cost is unclear for a real language like Java. Lazy evaluation has some nice properties for the execution time of attribute computations; only attributes that are being used add cost to the execution time. For example, attributes used to compute error messages do not add to the overall execution time as long as there are no errors in the analysed program. Similarly, an attribute can be defined for a large set of node types even though only a few nodes actually depend on the attribute value. This is useful for obtaining a simple specification, and yet does not incur additional cost. For example, references to primitive types can be broadcast throughout an entire AST, but will incur a computation cost only for the places where they are actually used.

Conditional rewriting is also done lazily and is triggered implicitly when a node is visited. When evaluating the condition for a rewrite, other nodes may be visited. This will in turn trigger further rewrites. When no conditions are true for rewrites in the visited nodes, the resulting tree node is returned. A nice property of this evaluation technique is that a traversal of the tree need not be aware of rewriting since the returned node is always in its final state. A thorough description of the evaluation of rewrites and its interaction with attribute evaluation is given in [16].

7. Related tools and systems

The current version of the JastAdd system builds loosely on an older version [23,22] which supported RAGs, inter-type declarations, and circular attributes. The current JastAdd system has added support for rewrites, parameterized attributes, broadcasting, and non-terminal attributes, thereby enabling the concise implementation of real programming languages like Java. The current system is also bootstrapped in itself.

7.1. Imperative systems

The primary application domain for the JastAdd system is extensible compilers and analysis tools. Our largest specification, the Java compiler, can be compared to handwritten extensible Java compilers. The Polyglot system [24, 25], is a Java 1.4 front-end supporting extending Java with new language constructs, translating them to Java source code. In contrast to JastAdd, the extensions in Polyglot are coded imperatively, making use of variants of the visitor design pattern. A phase-oriented architecture with fixed AST traversals is used, so that different computations need to be explicitly associated with different phases, and it is the burden of the user to make sure that everything is computed

in the appropriate order. Preliminary experiments with extending Java with AspectJ-like constructs indicate that using JastAdd leads to much more concise and clear specifications as well as to a faster translation tool.

JastAdd is used to specify contextual computations on top of an AST. The initial AST is usually built using a parser generator, e.g., JavaCC, ANTLR, CUP, or Beaver, using node constructors in semantic actions. Some of these tools support limited contextual computations through the use of visitors or semantic actions. Their imperative nature make these computations inferior to JastAdd modules from a modularity and extensibility point of view.

SmartTools is a semantic framework generator, based on XML and object technologies [26]. It combines generative approaches, model-driven architectures, and component technology to build entire tool suites. Their scope is clearly larger than ours but it is still interesting to compare the way languages are modelled and analysed. The main difference is the way semantic analysis is handled. SmartTools combine Visitors and AOP to provide extensible, reusable visitors that can be specialized to implement various analyses. While this approach is clearly superior to traditional visitors from a modularization point of view, they still have to manually schedule computations involving complex dependences.

7.2. Attribute grammar systems

There are many other attribute grammar systems, both commercial and licensed systems like the Synthesizer Generator [27] and Cocktail [28], as well as freely available systems like Eli [29], Elegant [30], LISA [31], LRC [32], and UAG [33]. While all of these systems support synthesized and inherited attributes, and many of them non-terminal attributes through higher-ordered grammars, there are many differences as compared to JastAdd.

One important difference is that most of them do not support reference attributes. One exception is the Elegant system that supports a notion similar to reference attributes which is used for name bindings, but via a global symbol-table data structure. Cocktail has a concept called tree-valued attributes which also seems similar to reference attributes, but we have not found any examples that show how they are used. In systems based on non-strict functional languages, like UAG, it should in principle be possible to use lazy evaluation to emulate reference attributes. However, we have not seen any documented examples that take advantage of this. In JastAdd, reference attributes constitute the key mechanism to deal with non-local dependences, not only for name bindings. AG systems that do not use reference attributes need to encode the context into attributes and pass them around explicitly, resulting in coupled specifications.

A very important difference between JastAdd and other systems is the support of context-dependent rewrites interleaved with attribute computations. This is a key mechanism that allows complex analysis problems like Java name resolution and type analysis to be broken down into small simple steps. To our knowledge, there are no other systems supporting similar mechanisms. Non-terminal attributes combined with forwarding [20] would be similar, but as far as we know, forwarding has only been implemented in prototypes built on top of Haskell, and it is unclear how the practical performance would scale to full languages like Java.

Another difference between JastAdd and the other AG systems mentioned above is the support for circular attributes and parameterized attributes. Previous work on circular attributes has been implemented in research systems of limited availability.

From a software engineering perspective there is an important difference between JastAdd and other AG systems in the integrated use of Java. The syntax for JastAdd specifications is a superset of Java, and constructs for attributes and equations are very Java-like, allowing users to think of attributes and equations as method declarations and method implementations. The aspect-oriented syntax used in JastAdd is very similar to that used in AspectJ. This is different from most other AG systems which use their own specific syntax for the attribution. This integration makes it straightforward to combine the declarative computations with imperative mainstream object-oriented programming in Java, and makes it easy for Java programmers to learn the tool and the concepts.

7.3. Transformation systems

The main focus of JastAdd is context-dependent computations on the AST, but since rewriting is also supported we also compare tree-transformation systems that are used for generating language-based tools, e.g., ASF+SDF [34], Stratego [35], and TXL [36].

An important difference between these systems and JastAdd is the way to specify the order of transformations. Transformation systems specify the order using implicit predefined traversals or explicit user-defined strategies.

JastAdd, on the other hand, uses fine-grained attribute dependences to drive the traversal which in turn implicitly defines the order of transformations. Complex traversal patterns can thus automatically follow dependences and do not have to be stated explicitly.

Another important difference is that transformation systems typically handle contextual information by using an external database that is updated during the transformations. This requires the user to explicitly associate database updates with particular transformation rules or phases. The traversal order must thus take contextual dependences, which can be highly non-local, into account. In contrast, JastAdd uses the contextual dependences to derive a suitable traversal strategy. The Stratego system has a mechanism for dependent dynamic transformation rules [37], supporting certain context-dependent transformations, but it is not clear how this could be used for implementing name binding and similar problems in object-oriented languages.

8. Conclusions

We have presented the JastAdd tool and shown how it supports implementation of extensible compiler tools and languages. A key design idea is to make use of declarative specification mechanisms in order to allow a high degree of decoupling between different modules, thereby supporting reuse and extensibility. Another key design idea is to build on object-orientation and Java, thereby both taking advantage of the support for modelling and reuse available in object-orientation, as well as making the declarative techniques easily understood by Java programmers.

In addition to well-known specification features like inherited, synthesized, and non-terminal attributes, JastAdd includes the very powerful feature of context-dependent rewrites, allowing the AST to be modified taking context-sensitive computations into account. A key feature is also that of reference attributes, allowing the AST itself to be used as the fact database. Additional JastAdd features like parameterized attributes and circular attributes also contribute substantially to the decoupling of modules and computations.

To demonstrate the full power of the tool we have successfully implemented a very strong case: a complete Java 1.4 compiler including compile-time checks and bytecode generation. Java 1.4 is a large complex language and implementing a complete compiler for it is a substantial undertaking, both because the language contains many idiosyncrasies that must be handled, and because it is an object-oriented language with many non-trivial constructs. We are not aware of any other declarative implementation of a complete practical object-oriented language. To demonstrate extensibility both for language constructs and tool functionality, we have extended the Java 1.4 compiler with new language constructs from Java 5, and extended the Java 1.4 front-end with devirtualization analysis. All these extensions have been done in a completely modular way.

Acknowledgements

We are grateful to Eva Magnusson for her work on circular attributes, to other researchers and students using the system, and to the anonymous reviewers for the valuable feedback and helpful comments.

Supplementary data

Supplementary data associated with this article can be found, in the online version, at doi:10.1016/j.scico.2007.02.003.

References

[1] T. Ekman, Extensible compiler construction, Ph.D. Thesis, Lund University, Sweden, June 2006.
[2] The Jacks compiler test suite. http://sources.redhat.com/mauve/, 2006.
[3] A. Nilsson, A. Ive, T. Ekman, G. Hedin, Implementing Java compilers using ReRAGs, Nordic Journal of Computing 11 (3) (2004) 213–234.
[4] Palpable Computing. http://www.ist-palcom.org, 2006.
[5] M. Fahndrich, K.R.M. Leino, Declaring and checking non-null types in an object-oriented language, in: Proceedings of OOPSLA'03, 2003, pp. 302–312.
[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, An overview of AspectJ, Lecture Notes in Computer Science 2072 (2001) 327–355.
[7] C. Clifton, G.T. Leavens, C. Chambers, T. Millstein, MultiJava: Modular open classes and symmetric multiple dispatch for Java, in: Proceedings of OOPSLA 2000, vol. 35 (10), 2000, pp. 130–145.
[8] D.E. Knuth, Semantics of context-free languages, Mathematical Systems Theory 2 (2) (1968) 127–145. Correction: Mathematical Systems Theory 5 (1) (1971) 95–96.

[9] G. Hedin, An object-oriented notation for attribute grammars, in: The 3rd European Conference on Object-Oriented Programming, ECOOP'89, in: BCS Workshop Series, Cambridge University Press, 1989, pp. 329–345.
[10] K. Koskimies, Object-orientation in attribute grammars, in: Proceedings on Attribute Grammars, Applications and Systems, Springer-Verlag, London, UK, 1991, pp. 297–329.
[11] G. Hedin, Reference attributed grammars, Informatica (Slovenia) 24 (3) (2000) 301–317.
[12] A. Poetzsch-Heffter, Prototyping realistic programming languages based on formal specifications, Acta Informatica 34 (1997) 737–772.
[13] J.T. Boyland, Descriptional composition of compiler components, Ph.D. Thesis, University of California, Berkeley, Available as Technical Report UCB//CSD-96-916, Sept. 1996.
[14] J.T. Boyland, Remote attribute grammars, Journal of the ACM 52 (4) (2005) 627–687.
[15] U. Kastens, W.M. Waite, Modularity and reusability in attribute grammars, Acta Informatica 31 (7) (1994) 601–627.
[16] T. Ekman, G. Hedin, Rewritable reference attributed grammars, in: Proceedings of ECOOP 2004, in: Lecture Notes in Computer Science, vol. 3086, Springer-Verlag, 2004, pp. 144–169.
[17] T. Ekman, G. Hedin, Modular name analysis for Java using JastAdd, in: Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal (Revised Papers), Lecture Notes in Computer Science, vol. 4143, Springer, 2006, pp. 422–436.
[18] H.H. Vogt, S.D. Swierstra, M.F. Kuiper, Higher order attribute grammars, in: Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, ACM Press, 1989, pp. 131–145.
[19] J. Saraiva, Purely functional implementation of attribute grammars, Ph.D. Thesis, Utrecht University, The Netherlands, 1999.
[20] E. Van Wyk, O. d. Moor, K. Backhouse, P. Kwiatkowski, Forwarding in attribute grammars for modular language design, in: Proceedings of CC 2002, in: Lecture Notes in Computer Science, vol. 2304, Springer-Verlag, 2002, pp. 128–142.
[21] R. Farrow, Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars, in: Proceedings of the SIGPLAN Symposium on Compiler Contraction, ACM Press, 1986, pp. 85–98.
[22] E. Magnusson, G. Hedin, Circular reference attributed grammars — their evaluation and applications, Electronic Notes in Theoretical Computer Science 82 (3) (2003).
[23] G. Hedin, E. Magnusson, JastAdd: An aspect-oriented compiler construction system, Science of Computer Programming 47 (1) (2003) 37–58.
[24] N. Nystrom, M.R. Clarkson, A.C. Myers, Polyglot: An extensible compiler framework for Java, in: Proceedings of CC 2003, in: Lecture Notes in Computer Science, vol. 2622, Springer-Verlag, 2003, pp. 138–152.
[25] A. Myers, N. Nystrom, X. Qi, Polyglot — a compiler front end framework for building Java language extensions. http://www.cs.cornell.edu/Projects/polyglot/, 2006.
[26] D. Parigot, C. Courbis, D. Rey, SmartTools Software Factories. http://http://www-sop.inria.fr/smartool/, 2006.
[27] GrammaTech, The synthesizer generator. http://www.grammatech.com/products/sg/, 2006.
[28] J. Grosch, Cocktail — compiler compiler toolkit Karlsruhe. http://www.cocolab.com/en/cocktail.html, 2006.
[29] A. Sloane, W.M. Waite, U. Kastens, Eli-translator construction made easy. http://eli-project.sourceforge.net/, 2006.
[30] Lex Augusteijn, The Elegant homepage. http://www.research.philips.com/technologies/syst%5fsoftw/elegant/, 2006.
[31] M. Lenic, E. Avdicausevic, D. Rebernak, M. Mernik, The LISA homepage. http://labraj.uni-mb.si/lisa/, 2006.
[32] M. Kuiper, D. Swierstra, M. Pennings, H. Vogt, J. Saraiva, Lrc: A purely functional, higher-order attribute grammar based system. http://www.di.uminho.pt/jas/Research/LRC/lrc.html, 2006.
[33] D. Swierstra, A. Baars, UAG — Utrecht attribute grammar system. http://www.cs.uu.nl/wiki/Center/AttributeGrammarSystem, 2006.
[34] M. van den Brand, P. Klint, The ASF+SDF MetaEnvironment. http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment, 2006.
[35] E. Visser, M. Bravenboer, R. Vermaas, Stratego: Strategies for program transformation. http://www.program-transformation.org/Stratego/WebHome, 2006.
[36] R. James, Cordy, TXL — Source transformation by example. http://www.txl.ca, 2006.
[37] K. Olmos, E. Visser, Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules, in: Proceedings of CC 2005, in: Lecture Notes in Computer Science, vol. 3443, Springer-Verlag, 2005, pp. 204–220.